Dustin Horvath
2729265
EECS 665
Lab 04
9/29/16

The purpose of this lab was to write a scanner/lexical analyzer for a C-like language of code. It would scan through all of the input, returning appropriate values for each identifier encountered, and outputting the TYPE for each token to stdout. It processes input directly from stdin, and so can simply be fed a file from the shell, and should support simple scripting.

A lexical analyzer works by reading through lines of input and processing the results sequentially. It generally does this by reading input a single character at a time, and comparing groups of symbols against rules or a grammar that describe the syntax of the programming language being analyzed. A lexical analyzer is a *state machine*. It cannot function and process complex code without having some memory of previously processed input. Coding elements such as comments can have long blocks of input spanning characters, whitespace, and newlines; elements such as these cannot be processed without a stateful machine.

As the scanner processes input, it reads a single character and checks it against its existing rules. If it matches a rule, it will perform the function listed inside the code block that corresponds, otherwise the scanner continues reading in additional characters until the list of symbols matches a rule in its list of declarations. This ensures that all text is read in sequence and nothing is omitted.

The program accompanying this report uses all the basic features of Lex. It has a block of symbols and regex definitions, rule declarations, and actions for every rule. One of the greatest concerns when processing input is *precedence*. Rules are evaluated in order, so care must be taken that the rule actions are ordered correctly. For the most part, the most specific rules need to be evaluated *first*, and more general rules must follow those.

Context/state-sensitive rules are also early in the list, including comments and string literals. At the very bottom of the rule list are the most catch-all, default type of rules. These include the rules for discarding whitespace, and the rule for identifying non-string-literal, non-identifier, non-special-symbol strings as “ID”; we must check for known words like “double” before strings like “my_instantiated_class”.

Special care was also taken with the “operations” group of rules. It was important that, for example, “==” be checked for prior to checking for “=”, because we don’t want the “==” symbol to be evaluated as two “SET” operations instead of the correct “EQ” operation.